

CS_232 Algorithms and Complexity

Week 11, Wednesday, 10/12/2008

Oliver Kullmann

Department of Computer Science
Swansea University
Swansea, SA2 8PP, UK

e-mail: O.Kullmann@Swansea.ac.uk

<http://cs.swan.ac.uk/~csoliver>

December 9, 2008

Lecture held on Wednesday, Dezember 10, 2008, in
the Wallace Building, Room 218, from 12:00 - 13:00.

I Revision lecture (examination preparation)

How to use the revision

- The purpose of this lecture (and the slides) is to emphasise the **main topics** of this course.
- It shall guide your exam preparation: When you go through the script and the courseworks, use the topics mentioned here to give more structure to the single lectures and to identify **recurrent themes**.
- And ask yourself (again and again; it takes time(!)) whether you really understand all the topics mentioned here.

Sometimes additional exercises are given — solving them will help you to understand better the subject, and to be prepared for the open questions in the examination.

Hopefully, when revising the lecture, you will have many “aha!”-experiences, where things become clear.

Overview

We had three main subjects:

1. graph theory and algorithms;
2. RSA (and supporting number theory);
3. complexity theory and algorithms.

The first subject can be split further into:

- graph reachability questions;
- hard graph problems (graph isomorphism and vertex colouring).

The main notions of graph theory

You must know

1. the definitions of **graphs**, **graphs with loops** and **general graphs**, both in formal terms and in your own words; what are the differences between these notions?
2. the difference between a (general) graph and a **graph drawing**;
3. the notions of **adjacency** and **incidence**;
4. the **line graph** $L(G)$;
5. the definition of **vertex degrees**;
6. when two (general) graphs are **equal** (and when not);
7. **subgraphs** and **induced subgraphs**;
8. the notion of a **graph isomorphism**, and when two (general) graphs are **isomorphic** (and when not).

Special graphs

You should know and understand the notions

1. K_n
2. $K_{n,m}$
3. $Gr_{n,m}$
4. P^k
5. C^k

and how to define and determine when a (general) graph is

1. **complete**
2. **bipartite**
3. **planar**
4. a **path graph**
5. a **cycle graph**.

Graph isomorphism

- The notion of **isomorphic graphs** is important.
- We discussed a **backtracking algorithm** in coursework I to determine whether two graphs are isomorphic or not. Understand this procedure and its complexity (always regarding time and space complexity).

Vertex degrees can help to distinguish between non-isomorphic graphs. Understand how this works.

Exercise: Find two graphs which have the same ordered sequence of vertex degrees, but which are not isomorphic.

Another nice exercise is to determine the **isomorphism types** of all connected graphs with up to five vertices.

And finally, you might consider for example cycle graphs, and what are the “essentially different” (i.e., non-isomorphic) possibilities to add one or two edges.

Notions related to reachability

You should be able to explain the following notions:

- **walk**
- **closed walk**
- **path** (neither edges nor vertices repeated)
- **circuit** (closed paths)

What is the **length of a walk**?!

When are two vertices **connected**?!

What are the **connected components** of a (general) graph?!

Forests and trees

The notions

- **forest**
- **tree**

are fundamental. And so is the notion of a **spanning tree** — you must be able to define these notions and to explain them in your own words.

Know the various **characterisations of trees!** (As always, if you have difficulties understanding them, then draw your own examples.)

Perhaps the most important characterisation of a tree is, that a general graph G with $n \in \mathbb{N}$ vertices is a tree if and only if G is connected and has (exactly) $n - 1$ edges.

“Abstract data types” for graphs

Know the possibilities to

- access graphs:
 - access to vertices (either to all vertices, or to all vertices adjacent to a given vertex)
 - access to edges (either to all edges, or to all edges incident to a given vertex, or to all edges connecting two given vertices);

- modify graphs:
 - adding vertices and edges;
 - deleting vertices and edges.

Know the two main data structures for representing graphs:

- adjacency lists

- adjacency matrices.

Graph traversal

You should be able to give pseudo-code for the **graph_traversal procedure**, and to explain the roles of the different inputs and outputs (especially the **marking object**, the **buffer object**, and the **visitor object**).

Understand the meaning of

- **back edges**
- **tree edges.**

Understand the **precise analysis** of graph_traversal procedure.

Understand how to **compute connected components** via graph_traversal procedure.

Understand how to **decide** via graph_traversal procedure whether a graph **is a tree or a forest**.

Rooted trees, BFS and DFS

Understand what **rooted trees** are (and what the difference is to “just trees”).

BFS (“breadth-first search”) and **DFS (“depth-first search”)** are very important strategies to apply the generic `graph_traversal` procedure. So you need to understand how they work, that is, first you need to understand the definitions, and second you must be able to apply it to examples. Again:

Create your own examples!

And third you should understand the **characteristic properties** of BFS-trees and DFS-trees (in these cases “tree” means “rooted tree”).

Distance problems: SPT

Understand what a **distance** in a graph is (for “normal” and for weighted graphs).

- Of course, you must know how to **compute distances** in a graph (that is, all edges have weight 1; the basic application of BFS).
- Know and understand **Dijkstra’s algorithm** for computing distances in weighted graphs with nonnegative weights (as a special case of `graph_traversal`). What is the complexity of this algorithm?

What about the algorithmic complexity of general distance problems (where edges can have negative weights)?!

Distance problems: MST

The second application of the generic `graph_traversal` to some sort of “metric problems” for graphs was the computation of **spanning trees of minimum weight** (“Prim’s algorithm”). Know how this works (and also try to get at least an intuitive feeling *why* it works).

A general remark here: **Don’t learn algorithms by memorisation!** Except of `graph_traversal`, algorithms haven’t been discussed in detail in the lecture, but instead we worked with so-called “higher-level descriptions”. These descriptions are of primary importance for our module.

(As I said, only the central algorithm `graph_traversal` needs to be known in detail.)

Vertex colouring

Of course, you must know the definition of a **vertex colouring** and of the **chromatic number** $\chi(G)$ of a graph.

Understand the **three assignment problems** we discussed, and how to **translate** them **into graph colouring problems**.

We considered three algorithms for graph colouring:

1. searching (“brute force”) for a k -colouring by running through all k^n possible maps;
2. the **greedy colouring algorithm**, using a given vertex ordering; when running through all vertex ordering (another form of “brute force” here), we have an algorithm finding an optimal colouring (and thus computing the chromatic number) in time $n!$;
3. the **traversing greedy colouring algorithm**, which runs in linear time (if the underlying `graph_traversal` runs in linear time), and

computes an optimal colouring in case the graph has a chromatic number at most three, but otherwise in general the colouring computed by this algorithm is not optimal.

Try to understand why the traversing greedy colouring algorithm correctly decides whether a graph is **bipartite** or not, or, in other words, why if the returned colouring uses three colours or more, then in fact at least three colours are needed.

Run the algorithm by hand on examples!

Exercise: Think about “edge colouring” and how it could be reduced to vertex colouring.

Cryptology

- Know how to encrypt and decrypt via **RSA**, and how to apply the “tools” needed here:
 - the **Euclidean algorithm** (simple and extended version);
 - **modular arithmetic**;
 - understand how to compute a multiplicative inverse in modular arithmetic by the Euclidean algorithm;
 - **fast exponentiation**;
 - **Euler’s phi-function**.
- Know the basic facts about the **security** of RSA.
- Understand the **basic principles** behind RSA.

Complexity theory

The absolute minimum is to know and understand the complexity classes **P**, **NP** and **co-NP**, and the meaning of **NP-completeness**.

It is important to understand the notions of a **decision problem** (informally and formally) and of **complexity measures**.

Furthermore you should have at least an intuitive understanding of the main **complexity classes** mentioned in the script.

Of course, for each algorithmic problem we considered in this module you should know its **current status**: P, NP, co-NP, NP-complete.

We also mentioned problems which are not in NP.

Also a basic understanding of the “**P vs. NP problem**” and the “**NP vs. co-NP problem**” is something you should have learned in this module.

The SAT problem

Know what the **SAT problem** is (in both forms, either using arbitrary propositional formulas, or only propositional formulas in **conjunctive normal form**).

Given simple examples, know how to decide their **satisfiability** resp. **unsatisfiability**.

What is a **clause-set** ? Know how to translate between conjunctive normal forms and clause-sets (and how to decide satisfiability of clause-sets).

The **reduction of colouring problems to SAT** we discussed is important for practical as well as theoretical reasons.

Also other translations into SAT are important. As exercise you might consider to translate logical puzzles into SAT problems.

Understand the **backtracking approach** to solve SAT problems.

Final remarks on the exam

Be as elaborate as possible (give me a chance to give you some marks(!)).

Study the courseworks (and the solutions) well.

It might be beneficial to get the latest versions of the script from the course home page.

Good luck! (Even better, if you don't need it.)

Merry Christmas and a Happy New
Year!

The End.